api-handling

Play+ API Service Helper Introduction ■ In the Play+ ecosystem, network communication is a foundational concern. The API layer is not just a series of fetch calls; it's a centralized, secure, and resilient gateway that ensures our applications communicate with the outside world predictably and reliably. This helper is based on the concept of a secure gateway pattern . It abstracts the complexities of network requests, so developers don't have to worry about authentication headers, error handling, or network resiliency. This aligns with our core design pillars by making the application more Adaptive to changing network conditions and more Intuitive for the developer, who can trust that security and best practices are handled automatically. By ensuring a reliable experience for users on all types of networks, it also supports our Inclusive pillar. Package Info ■ The Play+ API Service is a core part of the Golden Path starter kit. For existing projects, its functionality is typically bundled with the @playplus/security package. Description ■ Package / Path Description Golden Path (Recommended) Pre-installed (/system/api/) Uplift Path (Security Package) npm install @playplus/security Folder Reference ■ The API helper is composed of three cohesive files located within the /system/api directory, each with a distinct responsibility. File / Directory Purpose & Guidelines system/api/ Contains all core API handling logic. ■ apiProxy.ts The low-level engine. Handles fetch, headers, retries, and errors. Not used directly. ■ apiRoutes.ts The central route map. Defines all API endpoints, preventing magic strings. ■ apiService.ts The developer-facing interface. Provides get , post , etc., combining routes + proxy. config/ Can contain optional play.api.config.json for overriding default behaviors. Helper - Pillars Alignment ■ Pillar How This Helper Aligns Intuitive Abstracts immense complexity behind a simple, predictable interface (get , post), reducing cognitive load. Adaptive Automatically adapts to network failures with retries; handles dev/prod environments seamlessly. Inclusive Provides retries and caching to ensure usability even on slow or unstable networks. Helper Overview ■ The apiService is the developer's entry point for all network requests. It is an intelligent wrapper that automates the most difficult parts of API communication. The goal is to abstract the plumbing , so developers can focus on features—not boilerplate. It automates the following, completely in the background: Secure Token Injection : Attaches authentication and CSRF headers automatically. Intelligent Retries : Retries failed requests due to transient network or server errors. Timeouts : Prevents requests from hanging indefinitely. Error Normalization : Catches all network errors and standardizes them. Centralized Logging : Logs all request activity for observability. Smart Caching : Provides optional, automatic in-memory cache for GET requests. URL Management : Constructs full request URLs from environment variables and defined routes. Developers simply define their routes and call the appropriate method. The system handles the rest. Config Options ■ Global configuration can be provided in play.api.config.json . These values serve as defaults and can be overridden per-request. Config Variable Default Value Description Recommended Value timeoutMs 10000 Default timeout for all requests (ms). 10000 retry.maxAttempts 3 Max number of retry attempts for idempotent requests. 3 retry.delayMs 500 Base delay for the first retry, increases exponentially. 500 cache.defaultTtlMs 60000 Default TTL for cached GET requests (ms). 60000 Helper Methods ■ The apiService exposes intuitive methods for all common HTTP verbs. Method Name What It Does Method Signature get Performs a GET request. get<T>(route: string, options?: RequestOptions): Promise<T> post Performs a POST request. post<T>(route: string, body: any, options?: RequestOptions): Promise<T> put Performs a PUT request. put<T>(route: string, body: any, options?: RequestOptions): Promise<T> del Performs a DELETE request. del<T>(route: string, options?: RequestOptions): Promise<T> cached.get GET request with cache support if response is available. cached.get<T>(route: string, options?: CacheRequestOptions): Promise<T> RequestOptions allows overriding timeout, retry settings, or disabling auth with { auth: false } . Usage Examples ■ React: Fetching and Updating User Data ■ First, define your routes. This is done once and imported throughout the app. // system/api/apiRoutes.ts export const apiRoutes = { users : { getAll : "/users" , getById : (id : string) => ` /users/ ${ id } ` , update : (id : string) => ` /users/ ${ id } ` , } , auth : { login : "/auth/login" , } , } ; Then use apiService within your application logic: //

features/users/userService.ts import { apiService } from "../../system/api/apiService" ; import { apiRoutes } from "../../system/api/apiRoutes" ; import { User , UserCredentials } from "../types" ; export const userService = { async getAllUsers ( ) : Promise < User [ ] > { return apiService . get < User [ ] > (apiRoutes . users . getAll) ; } , async updateUser (id : string , data : Partial < User >) : Promise < User > { return apiService . put < User > (apiRoutes . users . update (id) , data) ; } , async login (credentials : UserCredentials) : Promise < { token : string } > { return apiService . post (apiRoutes . auth . login , credentials , { auth : false }) ; } , } ; Angular: User Service with Dependency Injection ■ // user.service.ts import { Injectable } from "@angular/core" ; import { apiService } from "@playplus/core" ; // Assuming it's provided import { apiRoutes } from "src/system/api/apiRoutes" ; import { User , UserCredentials } from "../models/user.model" ; import { Observable , from } from "rxjs" ; @ Injectable ({ providedIn : "root" }) export class UserService { getAllUsers ( ) : Observable < User [ ] > { return from (apiService . get < User [ ] > (apiRoutes . users . getAll)) ; } updateUser (id : string , data : Partial < User >) : Observable < User > { return from (apiService . put < User > (apiRoutes . users . update (id) , data)) ; } login (credentials : UserCredentials) : Observable < { token : string } > { return from (apiService . post (apiRoutes . auth . login , credentials , { auth : false })) ; } } Additional Info ■ Why We Created This Helper ■ Without a centralized API helper, every developer would need to manually handle critical cross-cutting concerns for every network request. This includes: Adding Authorization headers Implementing retry logic with exponential backoff Handling AbortController for timeouts Wrapping every call in try...catch and normalizing errors Managing cache and invalidation logic manually This approach is repetitive, error-prone, and leads to inconsistency. The apiService solves all of this in one secure, tested, centralized layer—offering a safe, productive foundation for all API communication. Developer Checklist ■ Have I added all new API endpoints to apiRoutes.ts ? Is API-calling logic kept out of UI components and encapsulated in services or hooks? Does the UI handle loading, error, and empty states gracefully? Have I created TypeScript types for API payloads (DTOs) and responses? For routes that don't require authentication, have I used the { auth: false } option? For rapid-fire events like search inputs, am I debouncing calls to the API service?